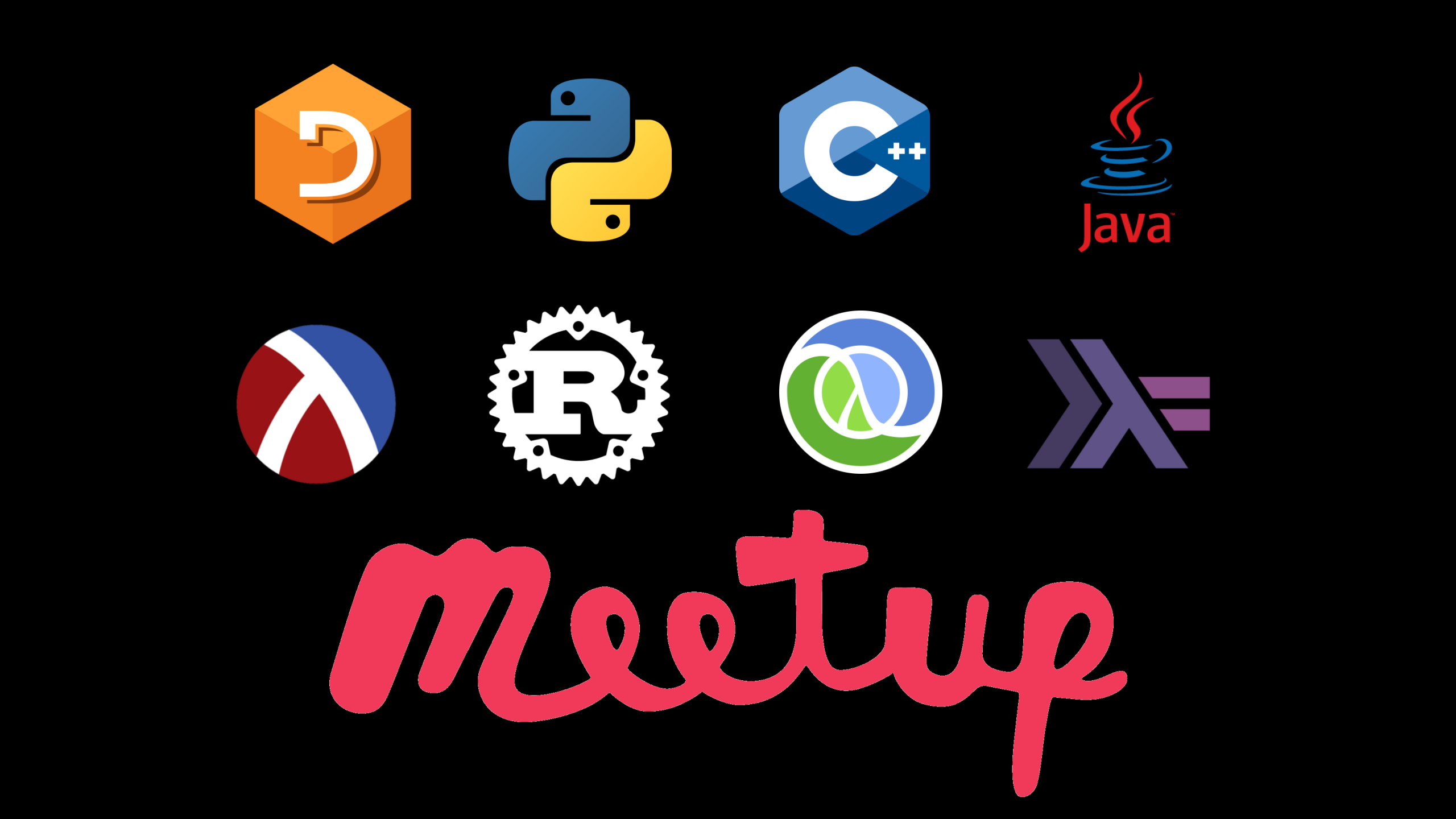
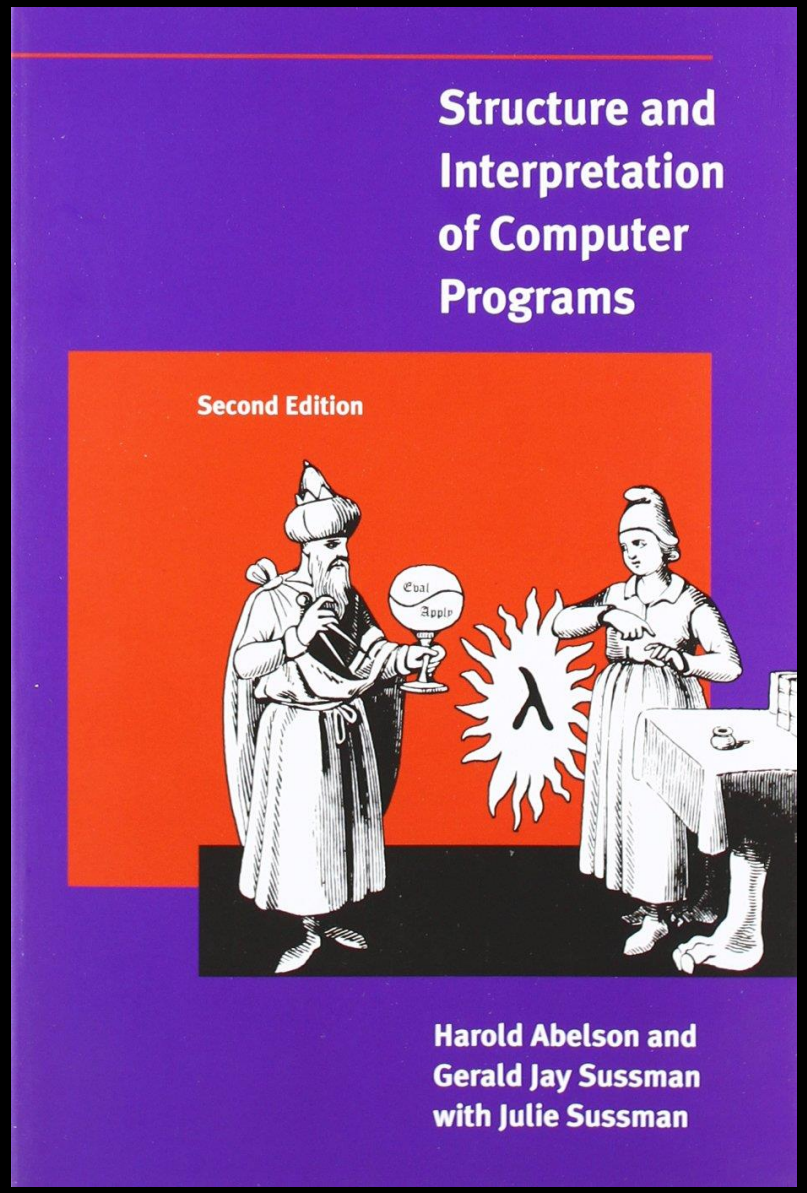

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 5.5

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

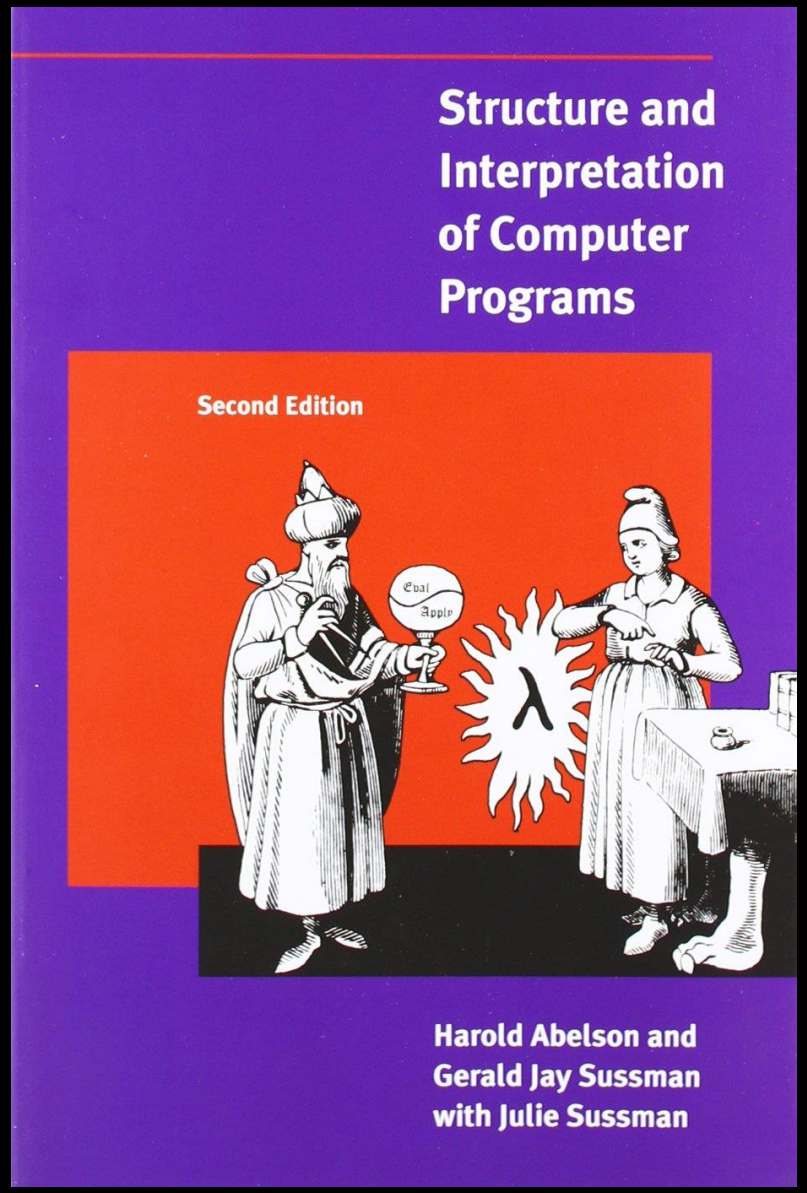
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 5.5

The explicit-control evaluator of Section 5.4 is a register machine whose controller interprets Scheme programs. In this section we will see how to run Scheme programs on a register machine whose controller is not a Scheme interpreter.

Compared with interpretation, compilation can provide a great increase in the efficiency of program execution, as we will explain below in the overview of the compiler. On the other hand, an interpreter provides a more powerful environment for interactive program development and debugging, because the source program being executed is available at run time to be examined and modified. In addition, because the entire library of primitives is present, new programs can be constructed and added to the system during debugging.

Interpreted

Compiled

This description suggests a strategy for implementing a rudimentary compiler: We traverse the expression in the same way the interpreter does. When we encounter a register instruction that the interpreter would perform in evaluating the expression, we do not execute the instruction but instead accumulate it into a sequence. The resulting sequence of instructions will be the object code. Observe the efficiency advantage of compilation over interpretation. Each time the interpreter evaluates an expression—for example, `(f 84 96)`—it performs the work of classifying the expression (discovering that this is a procedure application) and testing for the end of the operand list (discovering that there are two operands). With a compiler, the expression is analyzed only once, when the instruction sequence is generated at compile time. The object code produced by the compiler contains only the instructions that evaluate the operator and the two operands, assemble the argument list, and apply the procedure (in `proc`) to the arguments (in `argl`).

```
(define (compile exp target linkage)
  (cond ((self-evaluating? exp)
         (compile-self-evaluating exp target linkage))
        ((quoted? exp) (compile-quoted exp target linkage))
        ((variable? exp)
         (compile-variable exp target linkage))
        ((assignment? exp)
         (compile-assignment exp target linkage))
        ((definition? exp)
         (compile-definition exp target linkage))
        ((if? exp) (compile-if exp target linkage))
        ((lambda? exp) (compile-lambda exp target linkage))
        ((begin? exp)
         (compile-sequence
          (begin-actions exp) target linkage))
        ((cond? exp)
         (compile (cond->if exp) target linkage))
        ((application? exp)
         (compile-application exp target linkage))
        (else
         (error "Unknown expression type: COMPILE" exp))))
```

Targets and linkages

`compile` and the code generators that it calls take two arguments in addition to the expression to compile. There is a *target*, which specifies the register in which the compiled code is to return the value of the expression. There is also a *linkage descriptor*, which describes how the code resulting from the compilation of the expression should proceed when it has finished its execution. The linkage descriptor can require that the code do one of the following three things:

- continue at the next instruction in sequence (this is specified by the linkage descriptor `next`),
- return from the procedure being compiled (this is specified by the linkage descriptor `return`), or
- jump to a named entry point (this is specified by using the designated label as the linkage descriptor).

An instruction sequence will contain three pieces of information:

- the set of registers that must be initialized before the instructions in the sequence are executed (these registers are said to be *needed* by the sequence),
- the set of registers whose values are modified by the instructions in the sequence, and
- the actual instructions (also called *statements*) in the sequence.

Compiling linkage code

In general, the output of each code generator will end with instructions—generated by the procedure `compile-linkage`—that implement the required linkage. If the linkage is `return` then we must generate the instruction `(goto (reg continue))`. This needs the `continue` register and does not modify any registers. If the linkage is `next`, then we needn't include any additional instructions. Otherwise, the linkage is a label, and we generate a `goto` to that label, an instruction that does not need or modify any registers.[36]

[36]This procedure uses a feature of Lisp called *backquote* (or *quasiquote*) that is handy for constructing lists. Preceding a list with a backquote symbol is much like quoting it, except that anything in the list that is flagged with a comma is evaluated.

For example, if the value of `linkage` is the symbol `branch25`, then the expression

```
`((goto (label ,linkage)))
```

evaluates to the list

```
((goto (label branch25)))
```

Similarly, if the value of `x` is the list `(a b c)`, then

```
`(1 2 ,(car x))
```

evaluates to the list

```
(1 2 a).
```

```
(define (compile-linkage linkage)
  (cond ((eq? linkage 'return)
         (make-instruction-sequence '(continue) '()
          '((goto (reg continue)))))
        ((eq? linkage 'next)
         (empty-instruction-sequence))
        (else
         (make-instruction-sequence '() '()
          `((goto (label ,linkage)))))))
```

The linkage code is appended to an instruction sequence by `preserving` the `continue` register, since a `return` linkage will require the `continue` register: If the given instruction sequence modifies `continue` and the linkage code needs it, `continue` will be saved and restored.

```
(define (end-with-linkage linkage instruction-sequence)
  (preserving '(continue)
   instruction-sequence
   (compile-linkage linkage)))
```

```
(define (compile-self-evaluating exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,exp))))))
(define (compile-quoted exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '() (list target)
    `((assign ,target (const ,(text-of-quotation exp)))))))
(define (compile-variable exp target linkage)
  (end-with-linkage linkage
   (make-instruction-sequence '(env) (list target)
    `((assign ,target
              (op lookup-variable-value)
              (const ,exp)
              (reg env))))))
```

Compiling conditional expressions

The code for an `if` expression compiled with a given target and linkage has the form

```
⟨compilation of predicate, target val, linkage next⟩
 (test (op false?) (reg val))
 (branch (label false-branch))
true-branch
 ⟨compilation of consequent with given target
  and given linkage or after-if⟩
false-branch
 ⟨compilation of alternative with given target and linkage⟩
after-if
```

5.5.3 Compiling Combinations

The essence of the compilation process is the compilation of procedure applications. The code for a combination compiled with a given target and linkage has the form

```
⟨compilation of operator, target proc, linkage next⟩
⟨evaluate operands and construct argument list in argl⟩
⟨compilation of procedure call with given target and linkage⟩
```

```
(define (compile-application exp target linkage)
  (let ((proc-code (compile (operator exp) 'proc 'next))
        (operand-codes
         (map (lambda
                (operand) (compile operand 'val 'next))
              (operands exp))))
    (preserving '(env continue)
     proc-code
     (preserving '(proc continue)
      (construct-arglist operand-codes)
      (compile-procedure-call target linkage)))))
```

```
(compile
 '(define (factorial n)
    (if (= n 1)
        1
        (* (factorial (- n 1)) n)))
 'val
 'next)
```

Exercise 5.33: Consider the following definition of a factorial procedure, which is slightly different from the one given above:

```
(define (factorial-alt n)
  (if (= n 1)
      1
      (* n (factorial-alt (- n 1)))))
```

Compile this procedure and compare the resulting code with that produced for `factorial`. Explain any differences you find. Does either program execute more efficiently than the other?

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))


(define (factorial-alt n)
  (if (= n 1)
      1
      (* n (factorial-alt (- n 1)))))
```

```
;; resulting code of compiled procedure

((env)
 (val)
 ((assign val (op make-compiled-procedure) (label entry1) (reg env))
  (goto (label after-lambda2))
  entry1
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env (op extend-environment) (const (n)) (reg argl) (reg env))
  (save continue)
  (save env)
  (assign proc (op lookup-variable-value) (const =) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch6))
  compiled-branch7
  (assign continue (label after-call8))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
  primitive-branch6
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  after-call8
  (restore env)
  (restore continue)
  (test (op false?) (reg val))
  (branch (label false-branch4))
  true-branch3
  (assign val (const 1))
  (goto (reg continue))
  false-branch4
  (assign proc (op lookup-variable-value) (const *) (reg env))
  (save continue)
  (save proc)
  (save env)
  (assign proc (op lookup-variable-value) (const factorial-alt) (reg env))
  (save proc)
  (assign proc (op lookup-variable-value) (const -) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch9))
  compiled-branch10
  (assign continue (label after-call11))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
  primitive-branch9
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  after-call11
  (assign argl (op list) (reg val))
  (restore proc)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch12))
  compiled-branch13
  (assign continue (label after-call14))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
  primitive-branch12
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  after-call14
  (assign argl (op list) (reg val))
  (restore env)
  (assign val (op lookup-variable-value) (const n) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch15))
  compiled-branch16
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
  primitive-branch15
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
  after-call17
  after-if5
  after-lambda2
  (perform (op define-variable!) (const factorial-alt) (reg val) (reg env))
  (assign val (const ok))))
```

```
after-call8
(restore env)
(restore continue)
(test (op false?) (reg val))
(branch (label false-branch4))
true-branch3
(assign val (const 1))
(goto (reg continue))
false-branch4
(assign proc (op lookup-variable-value) (const *) (reg env))
(save continue)
(save proc)
(save env)
(assign proc (op lookup-variable-value) (const factorial-alt) (reg env))
(save proc)
```

```
34  (save proc)
35  (assign val (op lookup-variable-value) (const n) (reg env))
36  (assign argl (op list) (reg val))
37  (save argl)
38  (assign proc (op lookup-variable-value) (const factorial) (reg env))
39  (save proc)
```

```
34  (save proc)
35  (save env)
36  (assign proc (op lookup-variable-value) (const factorial-alt) (reg env))


37  (save proc)
```

```
64    after-call14
65    (restore argl)



66    (assign argl (op cons) (reg val) (reg argl))
```

```
62    after-call14
63    (assign argl (op list) (reg val))
64    (restore env)
65    (assign val (op lookup-variable-value) (const n) (reg env))
66    (assign argl (op cons) (reg val) (reg argl))
```

```
79  after-lambda2
80  (perform (op define-variable!) (const factorial) (reg val) (reg env))
81  (assign val (const ok))))
```

```
79  after-lambda2
80  (perform (op define-variable!) (const factorial-alt) (reg val) (reg env))
81  (assign val (const ok))))
```

```
34  (save proc)
35  (assign val (op lookup-variable-value) (const n) (reg env))
36  (assign argl (op list) (reg val))
37  (save argl)
38  (assign proc (op lookup-variable-value) (const factorial) (reg env))
39  (save proc)
```

```
34  (save proc)
35  (save env)
36  (assign proc (op lookup-variable-value) (const factorial-alt) (reg env))
37  (save proc)
```

```
64  after-call14
65  (restore argl)
66  (assign argl (op cons) (reg val) (reg argl))
```

```
62  after-call14
63  (assign argl (op list) (reg val))
64  (restore env)
65  (assign val (op lookup-variable-value) (const n) (reg env))
66  (assign argl (op cons) (reg val) (reg argl))
```

Exercise 5.38: Our compiler is clever about avoiding unnecessary stack operations, but it is not clever at all when it comes to compiling calls to the primitive procedures of the language in terms of the primitive operations supplied by the machine. For example, consider how much code is compiled to compute `(+ a 1)`: The code sets up an argument list in `argl`, puts the primitive addition procedure (which it finds by looking up the symbol `+` in the environment) into `proc`, and tests whether the procedure is primitive or compound. The compiler always generates code to perform the test, as well as code for primitive and compound branches (only one of which will be executed). We have not shown the part of the controller that implements primitives, but we presume that these instructions make use of primitive arithmetic operations in the machine's data paths. Consider how much less code would be generated if the compiler could *open-code* primitives—that is, if it could generate code to directly use these primitive machine operations. The expression `(+ a 1)` might be compiled into something as simple as[43]

```
(assign
 val (op lookup-variable-value) (const a) (reg env))
(assign val (op +) (reg val) (const 1))
```

```
;; Exercise 5.38 (page 814/5)

(compile
 '(+ a 1)
 'val
 'next)

;; resulting code of compilation

 ((env)
 (env proc argl continue val)
 ((assign proc (op lookup-variable-value) (const +) (reg env))
  (assign val (const 1))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const a) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch1))
  compiled-branch2
  (assign continue (label after-call3))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
  primitive-branch1
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  after-call3))
```

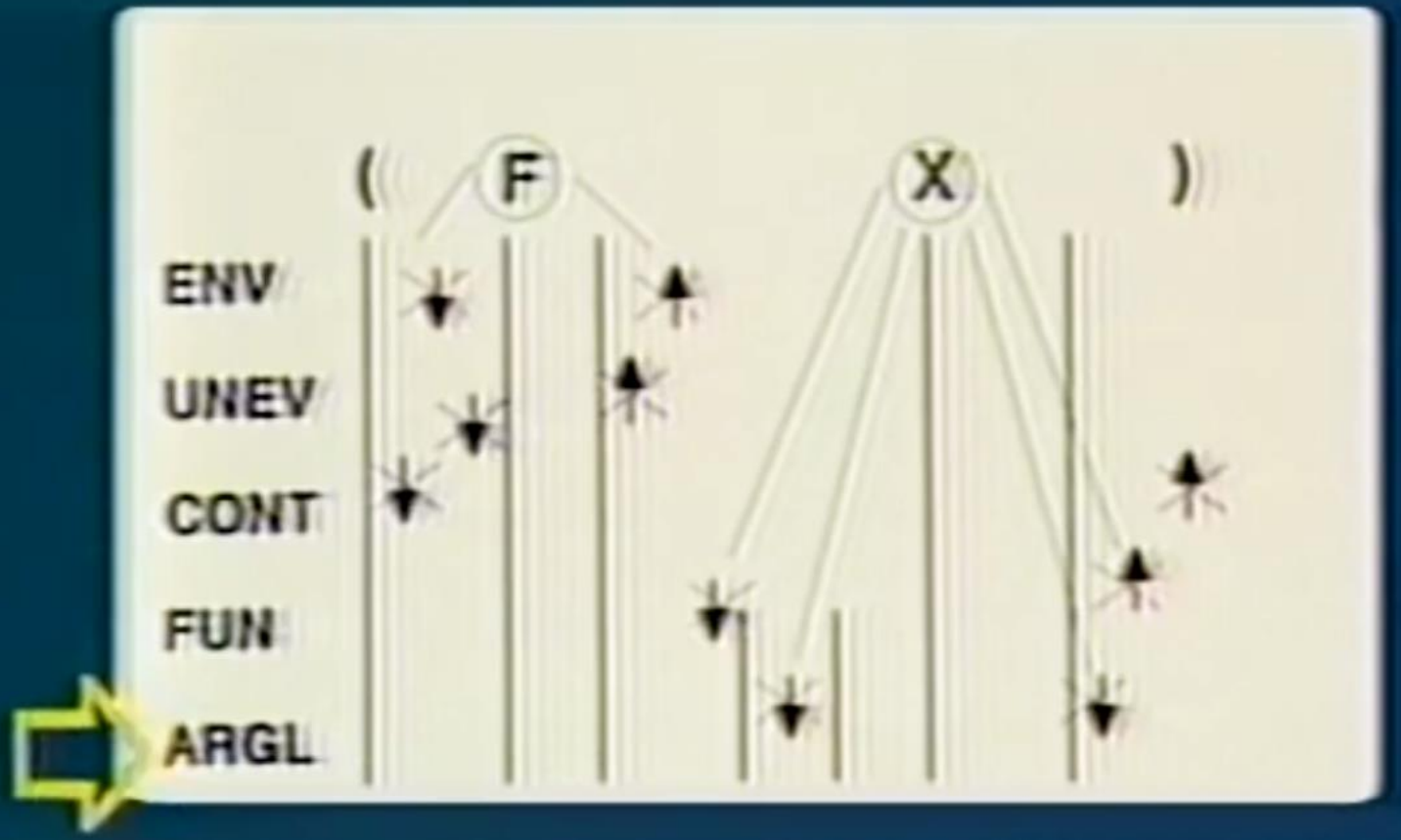
Structure & Interpretation of Computer Programs

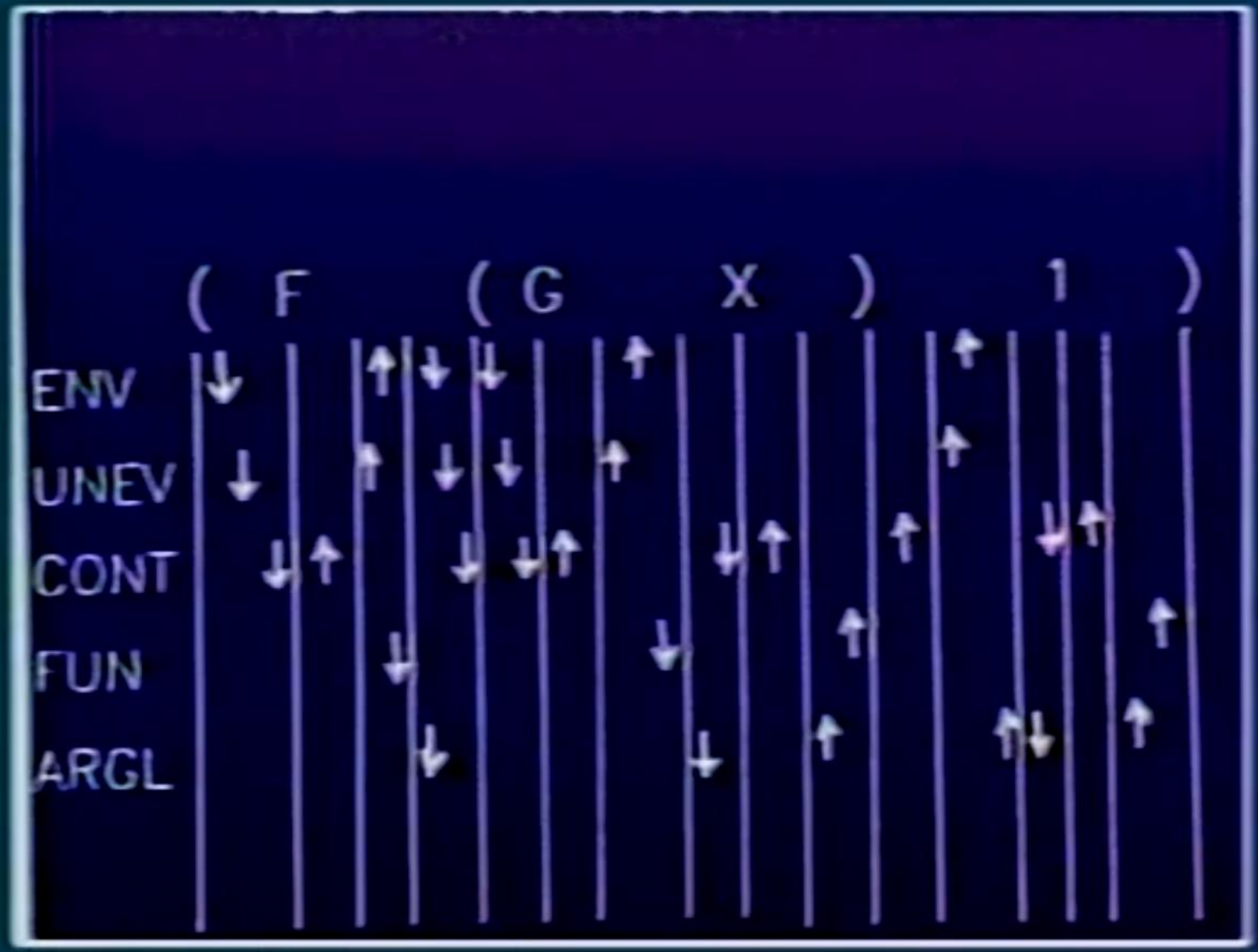
Harold Abelson

Gerald Jay Sussman

Register Operations in interpreting (F X)

```
(assign unev (operands (fetch exp)))
(assign exp (operator (fetch exp)))
(save continue)
(save env)
(save unev)
(assign continue eval-args)
(assign val (lookup-var-val (fetch exp) (fetc
(restore unev)
(restore env)
(assign fun (fetch val))
(save fun)
(assign argl '())
```


(
F
X
)
ENV
UNEV
CONT
FUN
ARGL

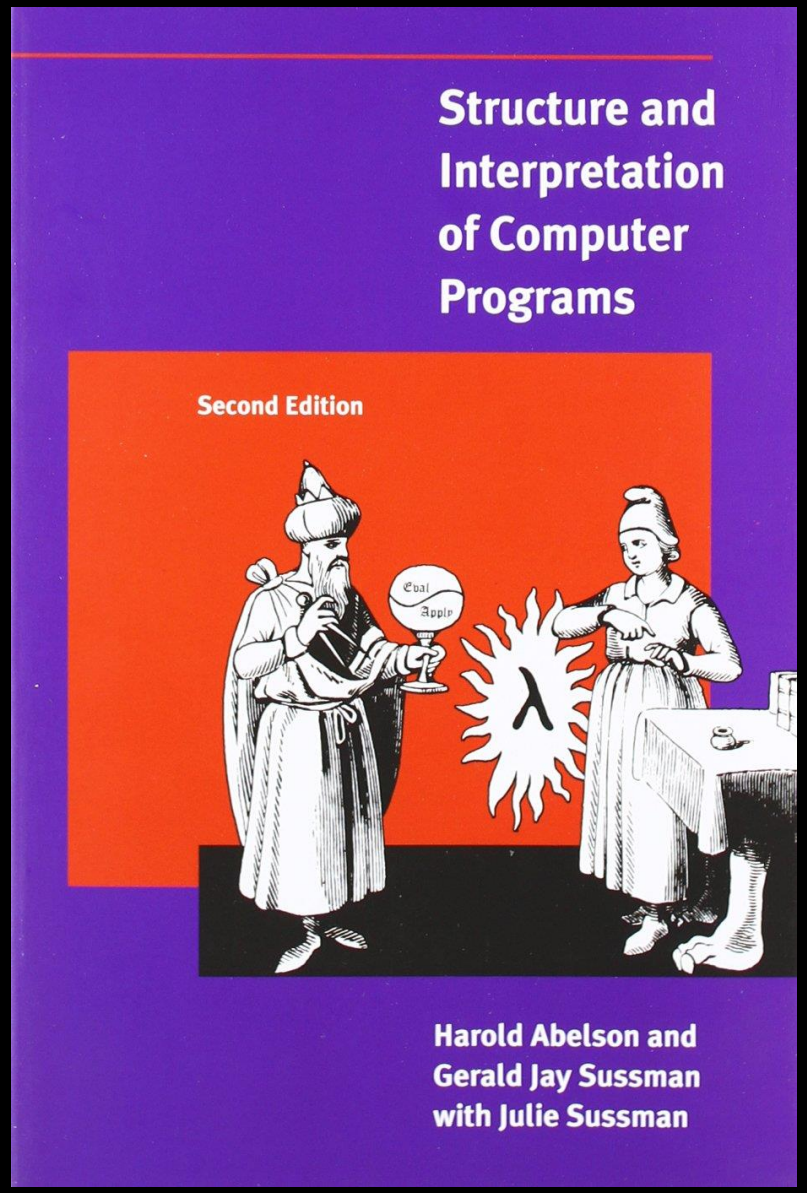

(F (G X) 1)
ENV
UNEV
CONT
FUN
ARGL


(F (G X) 1)
ENV
UNEV
CONT
FUN
ARGL

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman



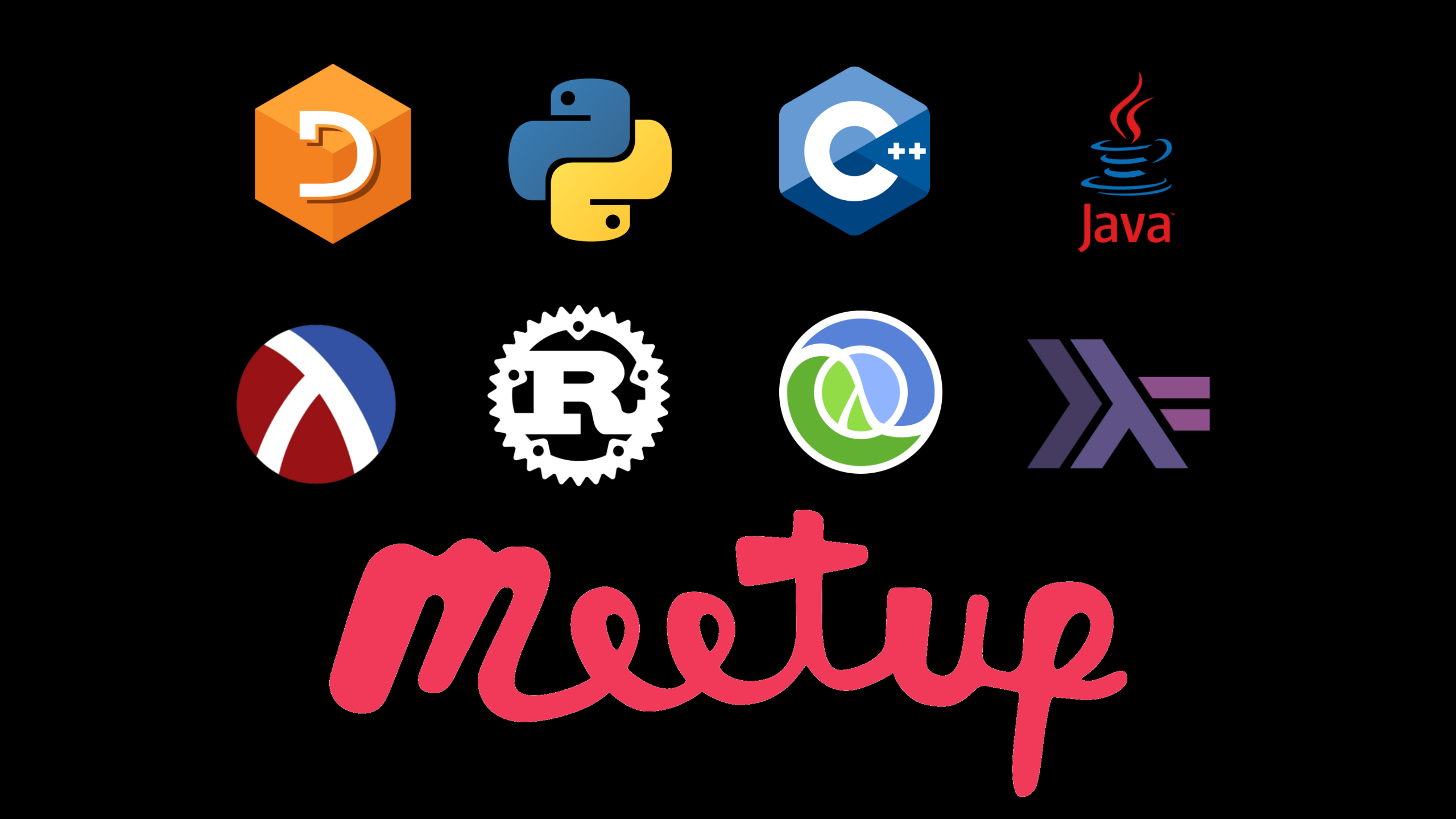
Structure and Interpretation of Computer Programs

Chaper 5.5

Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman


Java
meetup